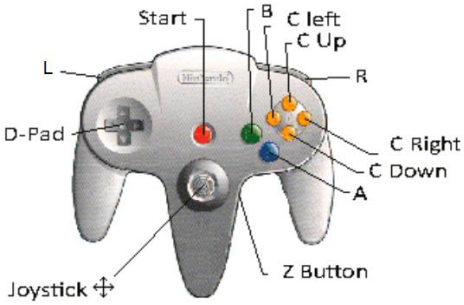
Design problem: Interfacing to a Nintendo 64 controller

In lab, your current task is to interface to an original Nintendo (NES) controller. Today we are going to take a look at interfacing to a Nintendo 64 (N64) controller.

How is this relevant? Why an N64 controller?



In lecture and lab we've mostly focused on the fundamentals: designing reasonable combinational logic, designing state machines and optimizing those things. That's reasonable given this is an introductory class—later classes will focus on applications. However, it's useful to see some of those applications. And in this case, we'll look at a nice interfacing problem—one you might see in 373 or elsewhere.

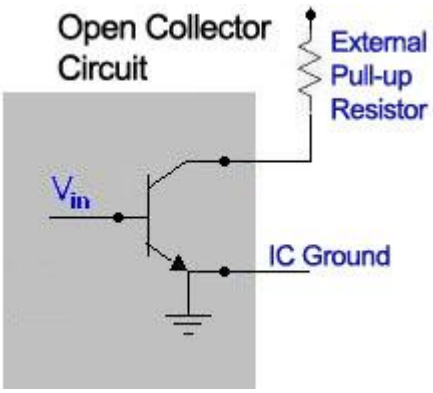
But why pick an N64 controller? Other devices, such as servos, ultrasonic sensors, and motors are much more common and therefore a more obvious choice. The big thing the N64 brings to the table is its level of complexity. Those other devices are all conceptually fairly simple to work with, though of course actual implementation details can be quite tricky. And more modern controllers, like the Xbox, use a much (much) more complex physical interface. The N64 is complex enough to pose a challenge and simple enough to cover in class.

Hardware considerations

First step is to understand the device and its hardware interfacing issues. The N64 has a total of 3 wires inside of the cable connecting the controller: Ground, Power and Data. Power is at about 3.3V.

Obviously Data is where all the interesting things happen. We send messages *both to and from* the controller using the Data wire. This could be done via tri-state devices. ***Design problem**: Show how we'd do that in the space below*.

However, the N64 controller doesn't do that. Rather, it uses an "open collector" scheme. In this case each device is either driving ground to the wire *or* is driving nothing (HiZ). The wire itself has a pull-up resistor so if no one is driving the wire low, it goes high. But if anyone is driving it low, it's low.



***Question**: This open-collector set-up emulates what type of gate?*

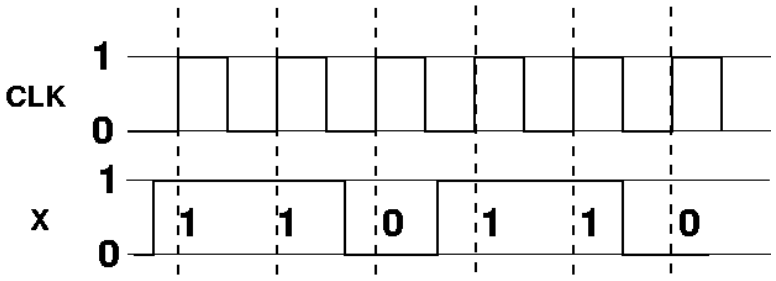
One of the biggest advantages of this setup is that we can't smoke things like we could with two devices using tri-state buffers. That's because we can't directly connect power to ground.

***Design problem:** Design a circuit which uses only standard gates and tristate devices. It takes three inputs: Dout, enable and bus. It has one output Din. Din should always be the value of the bus. If Enable=1, the output should be HiZ if Dout=1 and 0 if Dout=1.*

Data protocol

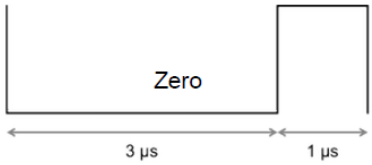
For any given device, there will be a communication protocol. That includes things like how to communicate 1s and 0s to who sends when, to what each message means. We'll look at the various issues one at a time, but with a focus on the first part—self-clocking.

Self-clocking

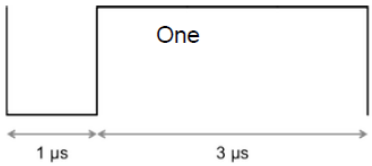


Normally, when we communicate between two devices we send data with a clock. Every device we've worked with has done this in one way or another. Consider our state machines—the input and output are all relative to the rising edge of the clock. For example, above we can tell that the data we are getting is 1, 1, 0, 1, 1, 0 because of the clock. Without the clock, we couldn't be certain exactly what was being sent. We might suspect the data is coming at twice the rate and see 1, 1, 1, 1, 0, 0, etc. (Each data element twice).

In order to save money and perhaps weight (yes, wires cost money) the fine folks at Nintendo choose to not send both data and a clock on two different wires. Instead they wanted to send them together. They could get away with this, because the controller doesn't need to send a lot of data—just the button statuses. And it doesn't need to send them quickly—one would assume 100Hz or so would be more than fast enough (10ms reaction time?)



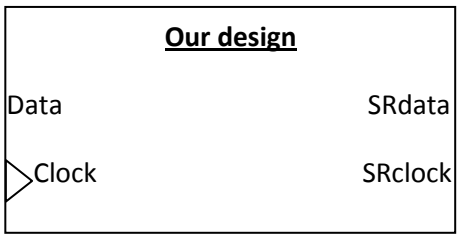
So, because there is only one wire for data, we need what is called a "self-clocked" signal. That is, the signal will send both clock and data. The scheme they chose to use is on the right.



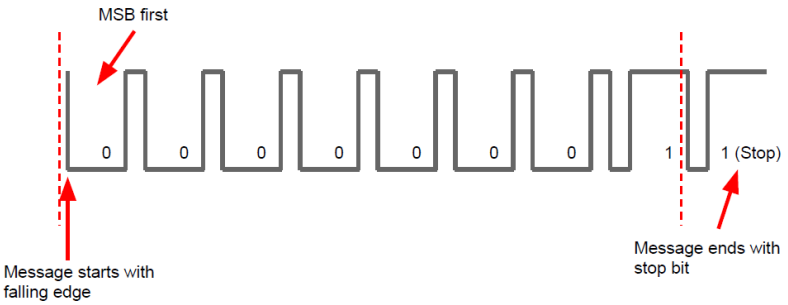
Question: *Why can't we just send data with an implicit clock? Say 1us? (This is a really tricky question!)*

Sending data in this way isn't all that hard. Basically we just need a very small state machine that gets passed a 1 or a 0 and it drives the output as needed. The only tricky part is that the output needs to be glitch free.

Receiving the data on the other hand, is quite tricky. We are going to spend the rest of class designing an interface to do exactly that. The first step is a high-level design. We are going to assume that we have "data" for input and a 25MHz clock. We'll assume our output is going to a shift register—we should generate a clock and data for that shift register.



From here, we'll do our work as a group and see what we come up with. Below are some useful figures.



0	A
1	B
2	Z
3	Start
4	Up
5	Down
6	Left
7	Right
8	"Joystick Reset"
9	(0)
10	L
11	R
12	C-Up
13	C-Down
14	C-Left
15	C-Right
16-23	X-Axis
24-31	Y-Axis
32	Stop bit (1)